Inventory Management System:

Importance of Data Structures and Algorithms in Inventory Management:

- Efficient inventory management involves handling large amounts of data, including product details, stock levels, transactions, and order history.
- Data structures and algorithms play a crucial role in achieving scalability, fast retrieval, and optimized operations.
- Efficient data structures allow **quick access** to inventory items, reducing search time and improving overall system performance.
- Properly designed data structures **minimize memory usage**, which is essential when dealing with extensive inventories.
- Algorithms help process **complex queries** (e.g., finding out-of-stock items, calculating reorder points) efficiently.
- **Well-structured** data prevents inconsistencies and ensures **accurate inventory tracking**.

Data Structures Suitable for Inventory Management:

- **HashMap:**
  - Ideal for fast lookups based on unique keys (e.g., product IDs).
  - Stores key-value pairs, where the key represents the product ID, and the value contains product details (name, quantity, price).
  - Provides constant-time (O(1)) access for retrieving product information.
  - Efficient for adding, updating, and deleting products.
- **ArrayList:**
  - Suitable for maintaining an ordered list of products.
  - Stores products in a linear array.
  - Allows efficient random access (O(1)) by index.
  - However, searching for a specific product requires linear time (O(n)).
- **Tree-based Structures:**
  - Useful when maintaining a sorted order of products (e.g., by name or price).

- Provides logarithmic time (O(log n)) for search, insertion, and deletion.
- Balances the trade-off between fast access and ordered storage.

Setup and Implementation:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package InventoryManagementSystem;

/**
 *
 * @author Aishwarya
 */
import java.util.*;

class Product {
    private int productId;
    private String productName;
    private int quantity;
    private double price;

    public Product(int productId, String productName, int quantity, double price) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }
public int getProductId() {
        return productId;
    }
```

```
 public String getProductName() {
        return productName;
    }

    public int getQuantity() {
        return quantity;
    }

    public double getPrice() {
        return price;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product ID: " + productId + ", Name: " + productName + ", Quantity: " +
quantity + ", Price: $" + price;
    }
}

public class InventoryManagementSystem {
    private HashMap<Integer, Product> inventory;

    public InventoryManagementSystem() {
        inventory = new HashMap<>();
```

```
    }

    public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
        System.out.println("Product added successfully.");
    }

    public boolean updateProduct(int productId, int newQuantity, double newPrice) {
        Product product = inventory.get(productId);
        if (product != null) {
            product.setQuantity(newQuantity);
            product.setPrice(newPrice);
            System.out.println("Product updated successfully.");
            return true;
        }
        System.out.println("Product not found.");
        return false;
    }

    public boolean deleteProduct(int productId) {
        Product removed = inventory.remove(productId);
        if (removed != null) {
            System.out.println("Product deleted successfully.");
            return true;
        }
        System.out.println("Product not found.");
        return false;
    }

    public void displayInventory() {
        if (inventory.isEmpty()) {
            System.out.println("Inventory is empty.");
```

```
            return;
        }
        for (Product product : inventory.values()) {
            System.out.println(product);
        }
    }

    public static void main(String[] args) {
        InventoryManagementSystem ims = new InventoryManagementSystem();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\nInventory Management System");
            System.out.println("1. Add Product");
            System.out.println("2. Update Product");
            System.out.println("3. Delete Product");
            System.out.println("4. Display Inventory");
            System.out.println("5. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter Product ID: ");
                    int productId = scanner.nextInt();
                    scanner.nextLine();
                    System.out.print("Enter Product Name: ");
                    String productName = scanner.nextLine();
                    System.out.print("Enter Product Quantity: ");
                    int quantity = scanner.nextInt();
                    System.out.print("Enter Product Price: ");
```

```
 double price = scanner.nextDouble();
                Product newProduct = new Product(productId, productName, quantity,
price);
                ims.addProduct(newProduct);
                break;

            case 2:
                System.out.print("Enter Product ID to update: ");
                productId = scanner.nextInt();
                System.out.print("Enter New Quantity: ");
                quantity = scanner.nextInt();
                System.out.print("Enter New Price: ");
                price = scanner.nextDouble();
                ims.updateProduct(productId, quantity, price);
                break;

            case 3:
                System.out.print("Enter Product ID to delete: ");
                productId = scanner.nextInt();
                ims.deleteProduct(productId);
                break;

            case 4:
                System.out.println("Current Inventory:");
                ims.displayInventory();
                break;

            case 5:
                System.out.println("Exiting...");
                scanner.close();
                return;
```

```
                default:
                    System.out.println("Invalid option. Please try again.");
            }
        }
    }
}
```

Time Complexity Analysis:

- **Add Operation**:
    - When adding a product to the HashMap, the time complexity depends on the hash function used to compute the key's hash value.
    - On average, adding an item to a HashMap has an expected time complexity of **O(1)** (constant time).
    - However, in rare cases (e.g., hash collisions), the complexity can be **O(n)** (linear time) due to probing or chaining.
- **Update Operation**:
    - Updating a product involves searching for it by its key (product ID) and modifying its attributes.
    - In a HashMap, the average time complexity for searching and updating is also **O(1)**.
    - Again, hash collisions can lead to worst-case scenarios with **O(n)** complexity.
- **Delete Operation**:
    - Deleting a product requires locating it by its key and removing it from the HashMap.
    - Similar to adding and updating, the average time complexity for deletion is **O(1)**.
    - Hash collisions may cause worst-case time complexity of **O(n)**.

Optimization Strategies:

- To optimize these operations:
    - **Load Factor Management**:
        - Monitor the load factor (ratio of filled slots to total slots) in the HashMap.

  - Resize the HashMap (rehash) when the load factor exceeds a threshold (e.g., 0.75).
  - This ensures efficient space usage and minimizes collisions.

- **Good Hash Function**:
  - Choose or design a good hash function to distribute keys uniformly across the hash table.
  - Avoid hash functions that cause clustering or poor distribution.
- **Open Addressing vs. Separate Chaining**:
  - If using open addressing (probing), consider linear probing or quadratic probing.
  - Experiment with different probing techniques to minimize collisions.
  - Alternatively, use separate chaining (linked lists) to handle collisions.
- **Resilience to Collisions**:
  - Implement a collision resolution strategy (e.g., double hashing, cuckoo hashing) to handle collisions gracefully.
  - Ensure that the chosen strategy doesn't degrade performance significantly.
- **Batch Operations**:
  - If you frequently perform multiple operations (e.g., batch updates), consider optimizing them together.
  - Group similar operations to minimize hash table resizing and rehashing.